Searching Patterns In Source Code

Detlef Streitferdt, Christian Heller, Ilka Philippow
Ilmenau Technical University
detlef.streitferdt@tu-ilmenau.de

Abstract

Maintaining software systems is a very time consuming activity, taking longer than actually developing the software. The crucial part within the maintenance phase is to understand the system. It is hard to understand legacy systems with poor or even no documentation. The recovery of an object-oriented software architecture is the first step towards understanding a system, but the resulting class structure is often still too complex to quickly get the idea of the system. The knowledge of design patterns possibly used can support a faster and better understanding of software systems. We evaluated existing pattern recognition approaches by the Information Retrieval criteria precision / recall and developed our own pattern search algorithms for the 23 patterns described in [1]. This paper presents first results of our approach for pattern search algorithms and discusses architectural issues of our implementation of the algorithms implemented as plug-in for the Together development IDE. This work is part of the InPULSE project [2], funded by the BMBF [3].

Keywords: Pattern Recognition, Software Patterns.

1 Introduction

Maintenance activities are aiming towards the management and integration of new or changed requirements. For this purpose software developers have to understand the existing system completely, although documentation like specifications or design models are poor or missing at all. Mostly only the source code, as the most rudimentary and reliable form of documentation, is available.

Design patterns offer predefined and tested solutions for fundamental design problems. The usage of design patterns leads to benefits for new and young developers by enabling them to reuse the knowledge of their experienced colleagues. Identification of design patterns contained in system as well as determination of source code classes for the identified patterns would lead to an improved understanding of the pattern based part of existing systems. Patterns are not explicitly described in software source code - excluding annotations or references in the documentation. The information about design patterns used in software systems is implicitly hidden and has to be detected manually in most cases.

This paper presents first results of our approach for pattern search algorithms in existing source code and discusses architectural issues of our implementation of the algorithms. Our approach is an extended version of existing pattern search algorithms based on minimal key structures. It focuses on the patterns described in [1], since they are a selection of practically relevant and useful patterns for software developers and the defacto standard. We present the first results of the search for patterns in four systems of a size of up to 1035 classes.

2 State-of-the-Art

Existing methods for automated pattern identification are evaluated according to the achieved results of their search algorithms. Depending on the found and actually existing patterns in a given system three results are possible:

- **True positive** in case a pattern has been recognized and the pattern is really implemented within the software system. This case is desired.
- **False positive** in case a pattern has been recognized and the pattern is not really implemented within the system. This case has to be avoided.
- **False negative** in case an implemented pattern has not been recognized. This case has to be avoided.

Based on the achieved results it is possible to derive metrics for the evaluation of searching tools, as described by the recall and the precision of the corre-

sponding algorithms. Both metrics are used widely for evaluating search results, e.g. in Information Retrieval [4].

- **Recall** is the number of all implemented patterns in a software system divided by the number of recognized patterns. A recall of 100% means that at least all implemented patterns were recognized. One might have recognized more, but the implemented patterns are all recognized - case 2, false positive has been avoided.
- **Precision** is the ratio of recognized and really implemented patterns (true positive) divided by the number of recognized patterns (sum of the results true positive and false positive). A precision of 50% means, that half of the recognized patterns are not really implemented in the software system.

Both values have to be taken into consideration for a tool evaluation. A precision value of 100% does not exclude false positive cases.

As the result of our evaluation of DP++ [5], KT [6], SPOOL [7], Pat [8], IDEA [9], the multi step search tool in [10], Fuzzy Logic algorithms [11], Pattern Wizard [12] and BACK-DOOR [13] we developed new algorithms and extended existing search algorithms by minimal key structures [14].

3 Searching minimal key structures

The novelty of our approach is the definition of *negative search criteria* for all of the 23 patterns described in [1]. With these criteria we developed search algorithms, which we used in our plug-in implementation for the Together IDE. In Figure 1 the graphical representation of expected, forbidden and uncertain elements within patterns is shown. To identify a part of a software system as pattern expected elements have to be part of this structure, forbidden elements are not allowed to be part of the structure and uncertain elements may or may not be part of the structure.

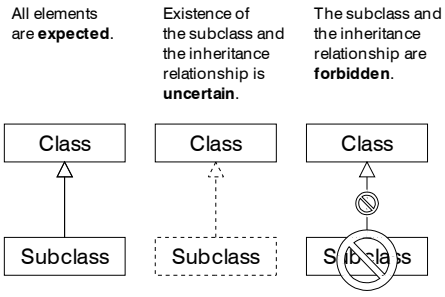

Figure 1: New search criteria

In case uncertain elements should be part of the analyzed system, all search criteria defined for these elements have to be checked as well. If the class `SpecializedAbstraction` in Figure 2 should appear in the system architecture, the class `Abstraction` is the mandatory super class of `SpecializedAbstraction`.

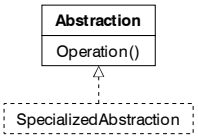

Figure 2: Example for uncertain criteria

As an example, developers can realize a variability point in the architecture of a system with the strategy pattern out of the behavioral group in Gamma. Many strategies can be implemented and even new strategies in future versions of the system are possible.

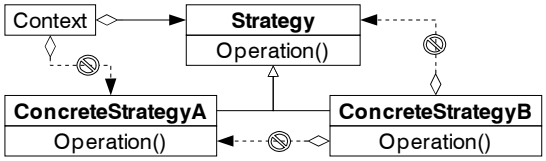

Figure 3: Search criteria for the strategy pattern

Our new search criteria are the basis for the algorithms we developed to analyze given source code. Thus, the code below addresses all the elements given in Figure 3.

```
create a set X of model-trees
for the given system

for-each tree T
{
 R = root-class of T
 if( R is abstract)
 {
  SC = subclasses of R
  for-all j in SC
  {
   if( (public interface subclass ==
        public interface root-class) &&
       j has no ref. to R &&
       j has no ref. to classes in SC)
   {
    for-all classes k in system
    {
     if(k has ref. to R &&
        k has no ref. to classes in SC )
     {
      ==> pattern found
     }
    }
   }
  }
 }
}
```

Pseudocode 1: Search algorithm for the strategy pattern

3.1 Plug-in Architecture

We implemented the pseudo-code of the 23 search algorithms, as shown in Pseudocode 1, in Java as a plug-in for the *Together v6.0.1* IDE (Integrated Development Environment). This implementation makes use of two APIs (Application Programming Interfaces) to access the reverse engineered UML (Unified Modeling Language) model of a software system and the corresponding source code, since not all the necessary information for pattern searching is available in the UML model, because of an incomplete reverse engineering. This situation is shown in Figure 4.

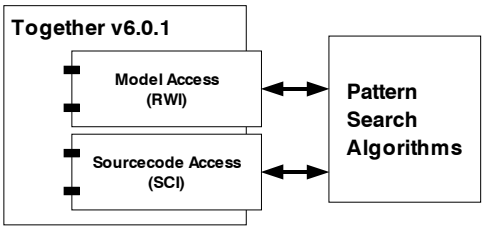

Figure 4: Architecture of the Plug-in

With our current implementation we encountered problems with source code leading to ambiguous UML models, as shown in Table 1. Especially the 0..* multiplicity is hard to analyze, given the numerous implementation possibilities for lists in C++ or Java.

Source Code	UML-Relation
B* ag;	A ◇— ag — B A ◇— 0..1 ag — B
B kp;	A ◆— kp — B
list<B> ag; vector<B> ag; deque<B> ag; OwnList ag;	A ◇— 0..* ag — B

Table 1: UML-Relation with multiplicities

In addition to the above mentioned ambiguities Gamma used the "creates" relationship which is not standardized in the UML. To realize all the algorithms we had to implement a rudimentary source code parser to extract code, that creates new classes like the `new` construct in C++.

3.2 Search Results

We tested our algorithms with several systems. *Patterns* is one of our own systems implemented in Java with 88 classes. It is the one-to-one implementation of all 23 Gamma patterns and has no functionality. Based on the fully known structure of the system with all its patterns, *Patterns* is currently our best test system. *Drawlet* is picture editor with 195 classes implemented by Rolemodel Software [15]. The Abstract Windowing Toolkit (AWT) is part of the Java 2 Standard edition [16] with 354 classes. As the fourth system we have taken *Tomcat* out of the *Jakarta* Open Source project [17] with 1035 classes.

The documentation taken from [18] for the *AWT* and *Tomcat* projects and taken from [19] for the *Drawlet* project was the only source to understand which patterns were used and where these patterns are located within the architecture of the software. Unfortunately the patterns are just sparsely documented. Thus, we were not able to calculate the values for *precision* and *recall* for these systems.

	System	*Patterns*		*Drawlet*		*AWT*		*Tomcat*	
	Search Algorithm	included	found	included	found	included	found	included	found
Creational	Abst. Fact.	1	1	n	3	y	10	n	4
	Builder	1	12	n	87	n	127	n	469
	Fact. Meth.	4	4	y	13	n	27	n	22
	Prototype	2	2	y	n	n	11	n	2
	Singleton	1	1	n	n	y	8	n	6
Structural	Adapter	1	3	n	40	n	52	y	123
	Bridge	1	25	n	61	y	61	n	345
	Decorator	1	1	n	n	n	n	n	n
	Facade	1	77	n	194	n	322	y	973
	Flyweight	1	1	n	n	y	n	n	n
	Composite	1	1	y	n	y	n	n	n
	Proxy	1	3	n	9	n	14	y	73
Behavioral	Command	1	1	n	13	n	20	n	46
	Observer	1	1	y	n	y	n	y	n
	Visitor	1	1	n	1	n	6	n	4
	Interpreter	2	2	n	n	n	n	n	n
	Iterator	1	1	y	n	n	n	n	n
	Memento	1	1	y	n	n	n	n	3
	Temp. Meth.	1	1	y	1	y	22	n	17
	Strategy	1	17	y	4	n	4	y	12
	Mediator	1	25	y	135	y	88	n	220
	State	1	1	n	n	n	n	n	2
	Chain Of R.	1	3	n	n	n	n	y	26
	The table entries are the number of patterns included in the system or found in the system. y Pattern was used when designing the system, but we don't know how often n We don't know wether the pattern is in the system.								

Table 2: Summary of our search results

The results for the *Patterns* system in Table 2 show, that 15 of the 23 algorithms are able to find patterns correctly. The remaining 8 algorithms have to be further analyzed to improve their searching results.

4 Conclusion and further work

In this paper we presented the first testing results of pattern search algorithms for the 23 patterns described by Gamma et al. in his book. The algorithms have been implemented in Java as plug-in for the Together IDE. The key point when searching patterns in source code is the quality of the reverse engineered model. Ambiguities between source code and UML models lead to incomplete class relations, which causes our algorithms to fail.

We are currently working on improved versions of our algorithms to gain a higher search precision. We are also working on an heavily extended reference implementation of all the Gamma patterns. With the full source code and documentation this reference system makes precise evaluation of pattern search algorithms possible, together with exact *precision* and *recall* values.

5 References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[2] InPULSE, "Integrative Pattern- und UML orientierte Lern- und System-Entwicklungsumgebung", 2004, www.inpulse-online.de.

[3] BMBF, "Bundesministerium für Bildung und Forschung", 2004, www.bmbf.de.

[4] Salton G., "Introduction to Modern Information Retrieval", McGraw-Hill, New York, 1983.

[5] Jagdish Bansiya, "Automatic Design-Pattern Identification", 1998, www.ddj.com/articles/1998/9806/9806a/9806a.htm?topic=patterns.

[6] Kyle Brown, "Design reverse-engineering and automated design pattern detection in Smalltalk", 1996, hillside.net/patterns/papers.

[7] Rudolf K. Keller, Reinhard Schauer, Sebastian Robitaille, Patrick Page, "Pattern-based reverse-engineering of design components", In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pp. 226-235, IEEE Computer Society Press, May, 1999.

[8] Christian Krämer, Lutz Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software", In Proceedings of the Working Conference on Reverse Engineering, pp. 208-215, Monterey, CA, November, 1996.

[9] Frederico Bergenti, Agostino Poggi, "Improving UML designs using automatic design pattern detection", In Proceedings of 12th International Conference on Software Engineering and Knowledge Engineering SEKE, pp. 336-343, Chicago, IL, 2000.

[10] G. Antoniol, R. Fiutem, L. Cristoforetti, "Design pattern recovery in object-oriented software", In Proceedings of the 6th International Workshop on Program Comprehension, pp. 153-160, Ischia, Italy, June, 1998.

[11] Jörg Niere, Jörg P. Wadsack, Lothar Wendehals, "Design pattern recovery based on source code analysis with fuzzy logic", 2001, www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2001/tr-ri-01-222.pdf.

[12] Hyoseob Kim, Cornelia Boldyreff, "A method to recover design patterns using software product metrics", In Proceedings of the 6th International Converence on Software Reuse ICSR6, Vienna, Austria, Juny, 2000.

[13] Forrest Shull, Walcelio L. Melo, Victor R. Basili, "An inductive method for discovering design patterns from object-oriented software systems", Technical Report UMIACS-TR-96-10, University of Maryland, 1991.

[14] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, Sebastian Naumann, "An approach for reverse engineering of design patterns", Software & Systems Modeling Journal, http://springerlink.metapress.com/link.asp?id=0dn4pmqh5uhnbk69, 2004.

[15] Rolemodel Software, "Homepage", 2004, www.rolemodelsoftware.com/drawlets/index.php.

[16] Sun.com, "Java 2 Standard Edition", 2004, java.sun.com/j2se/1.4.0.

[17] Apache.com, "Jakarta, Tomcat", 2004, jakarta.apache.org/tomcat.

[18] Wiki-Server, "PatternStories", 2004, wiki.cs.uiuc.edu/PatternStories/DesignPatterns.

[19] Ken Auer, "Fundamental Elements of an Extendible Java Framework", Rolemodel Software, www.rolemodelsoftware.com, 1997.